Q-OPT: Self-tuning Quorum System for Strongly Consistent Software Defined Storage[*]

Maria Couceiro
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
mcouceiro@gsd.inesc-id.pt

Gayana Chandrasekara
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
gayanaa@gmail.com

Manuel Bravo
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
angel.bravo@uclouvain.be

Matti Hiltunen
AT&T Labs Research
hiltunen@research.att.com

Paolo Romano
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
romano@inesc-id.pt

Luís Rodrigues
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
ler@tecnico.ulisboa.pt

ABSTRACT

This paper presents Q-OPT, a system for automatically tuning the configuration of quorum systems in strongly consistent Software Defined Storage (SDS) systems. Q-OPT is able to assign different quorum systems to different items and can be used in a large variety of settings, including systems supporting multiple tenants with different profiles, single tenant systems running applications with different requirements, or systems running a single application that exhibits non-uniform access patterns to data. Q-OPT supports automatic and dynamic reconfiguration, using a combination of complementary techniques, including top-k analysis to prioritise quorum adaptation, machine learning to determine the best quorum configuration, and a non-blocking quorum reconfiguration protocol that preserves consistency during reconfiguration. Q-OPT has been implemented as an extension to one of the most popular open-source SDS, namely Openstack's Swift.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms

Algorithms, Management, Performance

Keywords

Autonomic computing, quorum replication, machine learning

[*]This work was supported by the Fundação para a Ciência e Tecnologia (FCT) via the projects UID/CEC/50021/2013 and CMUP-EPB/TIC/0042/2013.

Middleware '15 December 07-11, 2015, Vancouver, BC, Canada
© 2015 ACM. ISBN 978-1-4503-3618-5/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814809.

1. INTRODUCTION

The advent of the cloud computing and of the software-defined data center paradigm have led to a drastic change to the way modern storage systems are engineered and managed. Modern cloud storage systems typically consist of a large number of commodity hardware components; this distributed storage infrastructure is virtualized and exposed as a service that can be provided to a single complex application, to multiple applications (that may or may not share data), or even to different customers with different requirements. Furthermore, with the emergence of the software-defined storage (SDS) model, sharing is supported by an interface that allows applications to quickly provision data storage (e.g., key-value stores[24] and personal file storages[14]) and to specify high level Service Level Agreements (SLAs) on performance, fault-tolerance, consistency, and durability.

Among the various mechanisms that are orchestrated by a SDS system (e.g., deduplication, security, backup and provisioning), replication is probably one of the most critical, given that fault-tolerance, and data availability, are fundamental requirements for most applications. Quorum systems are one of the most popular replication strategies in cloud-oriented/software-defined storage systems.

In a quorum-based system[21], the execution of a read or write operation on a data item requires to contact some subset, called a quorum, of the set of nodes that replicate that item [24, 11]. The choice of the size of the quorums used when executing a read or a write operation is one fundamental factor, affecting not only the consistency guarantees provided by the system, but also its performance and reliability. Quorum systems have several desirable properties that make them appealing: different choices of quorums can offer different consistency guaranties and, even for a given consistency level such as strong consistency, different quorum configurations may be selected, making the system highly tunable. Unfortunately, selecting the right quorum system is a non-trivial task. Given a consistency criteria, the number of quorum systems that can satisfy that criteria can be large, and different quorum systems provide different performance for different workloads.

Furthermore, different tenants, applications, or even a single application, may access different data items using distinct read/write ratios, and these access patterns are likely to change in time. For instance, a study on the utilization of Dropbox [14] provides concrete evidence of the existence of multiple and complex user patterns, where some users switch between periods characterized by write-intensive workloads and periods characterized by read-intensive, or even read-only, workloads (for instance, when users commute from office to home). Optimizing the configuration for several data-items is a complex, costly and error-prone task, which calls for automatic solutions that can react quickly to dynamic changes in the workloads.

This paper tackles this problem by proposing Q-OPT, a self-tuning system that, depending on the workload, dynamically adjusts the read and write quorum sizes of different items with the objective of maximizing a user-defined Key Performance Indicator, such as throughput or latency. Pursuing this goal requires tackling three main challenges: i) preserving the scalability of the system, by avoiding consuming too many resources with system monitoring or meta-data; ii) building accurate predictors for identifying the optimal configuration of the quorum sizes given the current workload; iii) designing mechanisms supporting the non-blocking reconfiguration of the quorum system, while preserving consistency during the transitioning between configurations.

Q-OPT copes with the scalability challenge by relying on lightweight, probabilistic top-k analysis to identify a restricted set composed by the mostly frequently accessed items. For this set of items, Q-OPT determines the optimal quorum size with a per-item granularity, whereas it adopts a coarse tuning granularity for the vast majority of infrequently accessed items, for which a common quorum configuration is used. Automatic tuning is performed by relying on state of the art machine learning (ML) techniques (a decision-tree classifier based on the C5.0 algorithm[34]), which are used to distil a predictive model for the expected optimal configurations of the read and write quorum sizes. This model is fed with a compact set of workload characteristics, which can be gathered efficiently via non-intrusive monitoring techniques. Consistency during the transition phases between different quorum configurations is preserved by means of a two-phase coordination algorithm, which allows the storage system to process incoming requests in a non-blocking fashion, i.e., while the reconfiguration is in process.

We integrated Q-OPT with Openstack's Swift[1], a popular open source SDS system. Our experimental data shows that the correct tuning of the quorum size can impact performance by up to 5x when using popular benchmarks, including YCSB[8]. We evaluate Q-OPT with more than 170 workloads and show that it achieves a throughput that is only slightly lower than when using the optimal configuration, incurring negligible throughput penalties during reconfigurations in most of the scenarios.

The remainder of this paper is structured as follows. Section 2 provides background on quorum systems and experimental data motivating the relevance of the problem tackled in this paper. Section 3 overviews Q-OPT. Sections 4, 5, and 6 provide a detailed description of the key components of Q-OPT. The experimental evaluation of Q-OPT is provided in Section 7. Finally, Section 8 discusses related work and Section 9 concludes the paper.

2. BACKGROUND AND MOTIVATIONS

2.1 Background on Quorums

Quorum systems are the cornerstone of a large number of techniques that address distributed coordination problems in fault-prone environments, including mutual exclusion, transaction processing, and byzantine fault-tolerance [19, 13, 24, 21, 27].

A *strict quorum system* (also called a coterie) is a collection of sets such that any two sets, called quorums, intersect [15]. In many distributed storage systems a further refinement of this approach is employed, in which quorums are divided into read-quorums and write-quorums, such that any read-quorum intersects any write-quorum (additionally, depending on the update protocol, in some cases any two write-quorums may also be required to intersect).

[1]`http://swift.openstack.org/`

There is a significant body of research that focused on which criteria should be used to select a "good" quorum configuration, which studied the trade-off between load, availability, and probe complexity [30, 33]. In practice, existing cloud-oriented storage systems, such as Dynamo [11], Cassandra [24] or OpenStack's Swift, opt for a simple, yet lightweight approach, according to which:

- each data item is replicated over a fixed set of storage nodes N, where N is a user configurable parameter (often called *replication degree*) that is typically much lower than the total number of the storage nodes in the system.
- users are asked to specify the sizes of the read and write quorums, denoted, respectively, as R and W. In order to guarantee strong consistency (i.e., the strictness of the quorums) these systems require that $R+W > N$.
- write and read requests issued by clients are handled by a *proxy*, i.e., a logical component that may be co-located with the client process or with some storage node. Proxies forward the read, resp. write, operation to a quorum of replicas, according to the current quorum configuration (i.e., reads are forwarded to R replicas and writes to W replicas). For load balancing, replicas are selected using a hash on the proxy identifier. A proxy considers the operation *complete* after having received the target R, resp. W, replies. If, after a timeout period, some replies are missing, the request is sent to the remaining replicas until the desired quorum is ensured (this fallback procedure, occurs rarely, mainly when failures happen).
- for read operations, after collecting the quorum of replies, a proxy selects the most recent value among the ones returned by the storage nodes. For write operations, storage nodes acknowledge the proxy but discard any write request that is "older" than the latest write operation that they have already acknowledged. These solutions assume that write operations are totally ordered, which is typically achieved either using globally synchronized clocks [26] or using a combination of causal ordering and proxy identifiers (to order concurrent requests), e.g., based on vector clocks [25] with commutative merge functions [11].

This scheme minimizes the load imposed on the system for finding a live read/write quorum in the good case given that, typically, the selected quorum of replicas that are contacted first are alive and do reply; only if faults occur, the requests need to be forwarded to all replicas. Further, thanks to the strictness property of the quorum system, and given the assumption on the total ordering of the write operations, this scheme guarantees that a read operation running solo (i.e., without concurrent write operations) returns the value produced by the last *completed* write operation.

Finally, the ability to tune the choice of the values of R and W opens the possibility to optimize the configuration of the quorum system to better match the workload characteristics, as we further discuss more in detail in the following section.

2.2 Impact of the Read/write Quorum Sizes

In this section we present the results of an experimental study aimed at quantifying the impact on performance of using different read and write quorum sizes in OpenStack Swift, a popular SDS solution. Before presenting the results, we briefly describe the experimental platform and methodology that we used.

Swift overview. Swift is the reference SDS of the OpenStack framework, a popular open source cloud (PaaS) platform. Swift is an object-oriented data store, and exposes a REST-ful API via

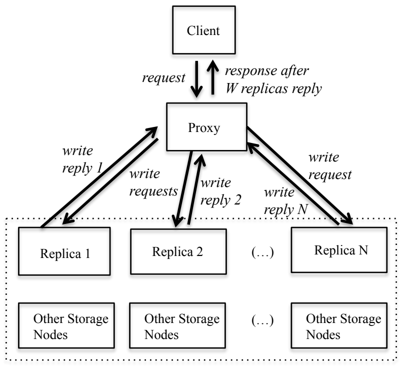

Figure 1: OpenStack's Swift Architecture.

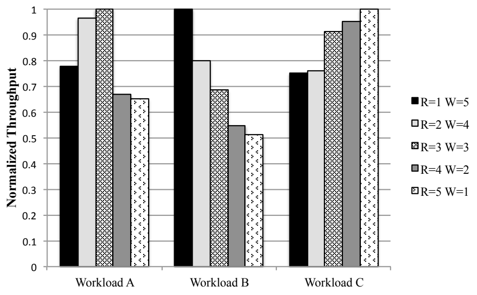

Figure 2: Normalized throughput of the studied workloads.

a set of proxy processes, which serve as intermediaries between the clients and the back-end storage nodes, as illustrated in Figure 1. As typical in many SDS systems, Swift allows users to define a wide range of static policies to isolate data (e.g., belonging to different tenants), as well as to control the replication strategies (e.g., total number of replicas of an object, and size of read/write quorums) and the mapping among objects and their physical location. In our experiments, we set the replication degree to 5 and use the default distribution policy that scatters object replicas randomly across the storage nodes (while enforcing that replicas of the same object are placed on different nodes).

Test-bed. The experimental test-bed used to gather the experimental results presented in the paper is a private cloud comprising a set of virtual machines (VMs) deployed over a cluster of 20 physical machines (1 VM per physical machine). The physical machines are connected via a Gigabit switch and each is equipped with 8 cores, 40 GB of RAM and 2x SATA 15K RPM hard drives (HDs). We allocate 10 VMs for the storage nodes, 5 VMs to serve as proxy nodes, and 5 VMs to emulate clients, i.e., to inject workload. Each client VM is statically associated with a different proxy node and runs 10 threads that generate a closed workload (i.e., a thread injects a new operation only after having received a reply for the previously submitted operation) with zero think time. Each storage VM runs Ubuntu 12.04, and is equipped with 2 (virtual) cores, 100GB disk and 9GB of RAM memory. On the other hand, each proxy and client VM is equipped with 8 (virtual) cores, 10GB disk and 16GB of RAM memory. In this motivating experiment we have used a single tenant and a workload where all objects are accessed with the same profile. In the evaluation of the full system we consider more complex scenarios with skewed non-uniform workloads.

Results. We start by considering 3 different workloads that are representative of different application scenarios. Specifically, we consider two of the workloads specified by the well known YCSB [8] benchmark (noted Workload A and B), which are representative of scenarios in which the SDS is used, respectively, to store the state of users' sessions in a web application, and to add/retrieve tags to a collection of photos. The former has a balanced ratio between read and write operations, the latter has a read-dominated workload in which 95% of the generated operations are read accesses. We also consider a third workload, which is representative of scenario in which the SDS is used as a backup service (noted Workload C). In this case, 99% of the accesses are write operations. Note that such write-intensive workloads are frequent in the context of personal file storage applications, as in these systems a significant fraction of users exhibits an upload-only access pattern [14].

Figure 2 shows the throughput of the system (successful operations per second) normalized with respect to the best read/write quorum configuration for each workload. These results were obtained using one proxy node and 10 clients. The results clearly show that when increasing the size of the predominant operation quorum, the number of served operations decreases: configurations favouring smaller read quorums will achieve a higher throughput in read-dominated workloads, such as Workload B, and vice-versa, configurations favouring smaller write quorums achieve a higher throughput in write-dominated workloads, such as Workload C. Mixed workloads such as Workload A, with 50% read and 50% write operations, perform better with more balanced quorums, favouring slightly reading from less replicas because read operations are faster than write operations (as these need to write to disk).

In order to assess to what extent the relation between the percentage of writes in the workload and the optimal write quorum size may be captured by some linear dependency, we tested approx. 170 workloads, obtained by varying the percentage of read/write operations, the average object size, and using 10 clients per proxy. In Figure 3 we show a scatter plot contrasting, for each tested workload, the optimal write quorum size and the corresponding write percentage. The experimental data clearly highlights the lack of a clear linear correlation between these two variables, and has motivated our choice of employing black-box modelling techniques (i.e., decision trees) capable of inferring more complex, non-linear dependencies between the characteristics of a workload and its optimal quorum configuration.

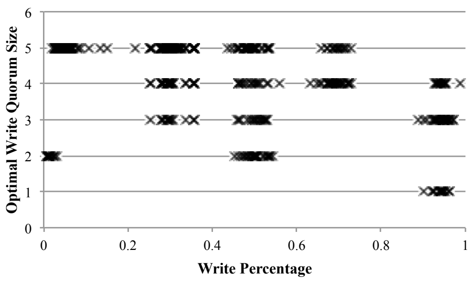

Figure 3: Optimal write quorum size vs write percentage.

3. Q-OPT OVERVIEW

Q-OPT is designed to operate with a SDS that adheres to the architectural organization discussed in Section 2, namely: its external interface is represented by a set of proxy agents, and its data is distributed over a set of storage nodes. We denote the set of proxies by $\Pi = \{p_1, \ldots, p_P\}$, and the set of storage nodes by $\Sigma = \{s_1, \ldots, s_S\}$. In fact proxies and storage nodes are logical process which may be in practice mapped to a set of physical nodes using different strategies (e.g., proxies and storage nodes may reside in two disjoint sets of nodes, or each storage node may host a proxy). We assume that nodes may crash according to the fail-stop (non-byzantine) model. Furthermore, we assume that the system is asynchronous, but augmented with unreliable failure detectors which encapsulate all the synchrony assumptions of the system. Communication channels are assumed to be reliable, therefore each message is eventually delivered unless either the sender or the receiver crashes during the transmission. We also assume that communication channels are FIFO ordered, that is if a process p sends messages $m1$ and $m2$ to a process q then q cannot receive $m2$ before $m1$.

As illustrated in Figure 4, Q-OPT is composed of three main components: the Autonomic Manager, the Reconfiguration Manager, and the Oracle, that we briefly introduce below.

The *Autonomic Manager* is responsible for orchestrating the self-tuning process of the system. To this end, it gathers workload information from the proxy processes, and triggers reconfigurations of the quorum system. The optimization logic of the Autonomic Manager is aimed at maximizing/minimizing a target KPI (like throughput or latency), while keeping into account user defined constraints on the minimum/maximum sizes of the read and write quorums. This allows, for instance, for accommodating fault-tolerance requirements that impose that each write operation to contact at least $k > 1$ replicas. The Autonomic Manager is prepared to collect statistics for different data items and to assign different quorums to different objects. However, for scalability issues, we avoid to collect individual statistics for all objects. In fact, Q-OPT makes a fine grain optimization for the objects mostly accessed, and that consume the largest fraction of the system resources and then treats in bulk all objects that are in the tail of the access distribution.

More precisely, Q-OPT starts to perform multiple rounds of fine-grain optimization, that works as follows. In each round, a top-k analysis is performed to i) identify the (next) top-k objects that have not been optimized yet, and ii) monitors the access to the top-k objects (identified in the previous round) and extract their read-write profile. Then, the read/write quorum for those objects is optimized (see more detail below). At the end of each round, the changes are applied and the effect of these, on the system performance, is checked. If the average performance improvement over the last γ rounds is above a predefined threshold θ, a new round of fine-grain optimization is performed (for the next top-k objects). When the improvement achieved with the fine-grain optimization is below the threshold, the fine-grain optimization of the system is considered to be terminated. At this point, the system uses the average information collected for the remaining objects (which are on the tail of the access frequency distribution) and a read/write quorum is selected for all those objects based on their aggregated profile.

For optimizing the access to a given data item (or to an aggregated set of data items), the Autonomic Manager relies on an *Oracle* that encapsulates a black-box machine-learning based predictor that is responsible for determining the best quorum, given the workload of the monitored SDS application.

Finally, the *Reconfiguration Manager* coordinates a non-blocking reconfiguration protocol that aims at altering the read/write quorum size used by the proxies for a given data item. The reconfiguration is non-blocking in the sense that it does not require halting the processing of read/write operations during the transition phase from the old to the new configuration, even in presence of faults affecting some proxies and/or the Reconfiguration Manager. The reconfiguration protocol is oblivious to the specific quorum-based protocol used to replicate data in the SDS (which may, e.g., provide the semantics of regular or atomic register[18]), but it guarantees that at any time (i.e., before, during, and after the

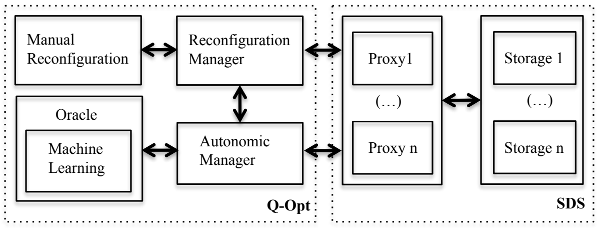

Figure 4: Architectural overview.

reconfiguration process) the quorum used by a read operation intersects with the write quorum of concurrent write operations, or, in absence of concurrent writes, of the last completed write operation.

A detailed description of each of these three components is provided in the following sections. Note that, for convenience of conciseness, in the remaining of the paper we describe the three components above as if they have centralized, non fault-tolerant, implementations. However, standard replication techniques, such as state-machine replication [18, 38, 5], can be used to derive fault-tolerant implementations of any of these components, such that they not become single points of failure in the system.

4. QUORUM OPTIMIZATION

Three key components are involved in the quorum optimization process: the proxies, the autonomic manager, and the oracle.

The Autonomic Manager is responsible for determining when to trigger a quorum reconfiguration. The pseudo code executed at the Autonomic Manager side is depicted in Algorithm 1. It executes the following tasks:

It first starts a new round by broadcasting the new round identifier, r, to all the proxies (line 5). Each proxy p_i then replies with (line 7):

- $topK_i^r$: A set of new "hotspots" objects that, according to the proxy's local accesses, should be optimized in the next round to obtain larger benefits. In order to be able to identify the "hotspots" on each proxy with low overhead, Q-OPT adopts a state of the art stream analysis algorithm [28] that permits to track the top-k most frequent items of a stream in an approximate, but very efficient manner.
- $statsTopK_i^{r-1}$: The ratio of write accesses and the size for each of the objects resulting in the top-k analysis of the previous round.
- $statsTailK_i^{r-1}$: Aggregate workload characteristics (see Section 6 for detailed information on the traced workload characteristics) for the objects whose quorum size has not been individually optimized, i.e., the objects in the tail of the access distribution.
- th_i: The throughput achieved by the proxy during the last round.

Once the information sent by the proxies is gathered and merged (line 8 and 9), the merged statistics of the previous round is fed as *input features* to the Oracle (line 10), that outputs a *prediction* (line 11) of the quorum to use *for each* object in the top-k set (see details in Section 6). In the current prototype, the Oracle only outputs the size W of the write quorum and the size R of the read quorum is derived automatically based on the system's replication degree, i.e., $R = N - W + 1$. If the output of the Oracle is different from the current quorum system for that object, a reconfiguration is triggered. In this case, the Autonomic Manager interacts with the Reconfiguration Manager (lines 12 and 13), which is in charge of orchestrating the coordination among proxy and storage nodes and adapt

```
1  int r=0; // Round identifier
2  // Fine-grain round-based optimization.
3  do
4      r=r+1;
5      broadcast [NEWROUND, r] to $\Pi$;
6      $\forall p_i \in \Pi$ :
7          wait received [ROUNDSTATS, r, $topK_i^r$, $statsTopK_i^{r-1}$, $statsTail_i^{r-1}$, $th_i^{r-1}$] from $p_i \vee suspect(p_i)$;
8      $statsTopK^{r-1}$=merge($statsTopK_1^{r-1}$, .., $statsTopK_P^{r-1}$);
9      $topK^r$=merge($topK_1^r$,...,$topK_P^r$);
10     send [NEWSTATS, r, $statsTopK^{r-1}$] to ORACLE;
11     wait received [NEWQUORUMS, r, $quorumsTopK^{r-1}$] from ORACLE;
12     send [FINEREC, r, $\langle topK^{r-1}$, $quorumsTopK^{r-1}\rangle$] to RM;
13     wait received [ACKREC, r] from RM;
14     broadcast [NEWTOPK, r, $topK^r$] to $\Pi$;
15     $th^{r-1}$=aggregateThroughput($th_1^{r-1}$,..., $th_P^{r-1}$) ;
16     $\Delta_{th}(\gamma)$ = throughput increase over last $\gamma$ rounds.;
17 while $\Delta_{th}(\gamma) \geq \theta$
18 // Tail optimization.
19 $statsTail^{r-1}$=merge($statsTail_1^{r-1}$, ..,$statsTail_P^{r-1}$);
20 send [TAILSTATS, $statsTail^{r-1}$] to ORACLE;
21 wait received [TAILQUORUM, $quorumTail^{r-1}$] from ORACLE;
22 send [COARSEREC, $quorumTail^{r-1}$] to RM;
23 wait received [ACKREC, r] from RM;
```

Algorithm 1: Autonomic Manager pseudo-code.

the current quorum configuration for the top-k objects identified in the previous round. Otherwise, if the current configuration is still valid, no reconfiguration is triggered. As a final step of a fine-grain optimization round, the Autonomic Manager broadcast the current top-k set to the proxies. Thus, each proxy can start monitoring the objects that belong to the current top-k set in the next round.

At the end of each round, the Autonomic Manager, based on the average throughput improvements achieved during the last γ rounds, decides whether to keep optimizing individual objects in a fine-grain manner or to stop. When the gains obtained with the fine-grain optimization of individual "hotspot" objects becomes negligible (i.e., lower than a tunable threshold θ), a final optimization step to tune the quorum configurations used to access the remaining objects, i.e., the objects that fall in the tail of the access distribution. These objects are treated as bulk (lines 19 - 22): the same read/write quorum is assigned to all the objects in the tail of the access distribution based on its aggregate workload characterization.

The frequency with which the Autonomic Manager cycle is executed is regulated by a classical trade-off for any autonomic system: the more often the Autonomic Manager queries the machine learning model, the faster it reacts to workload changes. However, it also increases the risk to trigger unnecessary configuration changes upon the occurrence of momentary spikes that do not reflect a sustained change in the workload. In our current prototype we use a simple approach based on a moving average over a window time of 30 seconds, which has proven successful with all the workloads we experimented with. As with any other autonomic system, in our implementation there is also a trade-off between how fast one reacts to changes and the stability of the resulting system. In the current prototype, we simple use a fixed "quarantine" period after each reconfiguration, to ensure that the results of the previous adaptation stabilise before new adaptations are evaluated. Of course, the system may be made more robust by introducing techniques to filter out outliers [20], detect statistically relevant shifts of system's metrics [32], or predict future workload trends [22].

5. RECONFIGURATION MANAGER

The Reconfiguration Manager (subsequently denoted RM) executes the required coordination among proxy and server nodes in order to allow them to alter the sizes of read and write quorums without endangering neither, consistency, nor availability during reconfigurations. This coordination enforced by Q-OPT is designed to preserve the following property, which is at the basis of all quorum systems that provide strong consistency:

Dynamic Quorum Consistency. *The quorum used by a read operation intersects with the write quorum of any concurrent write operation, and, if no concurrent write operation exists, with the quorum used by the last completed write operation.*

where two operations o_1, o_2 are concurrent if at the time in which a proxy starts processing o_2, the processing of o_1 by a (possibly different) proxy has not been finalized yet (or vice-versa). The availability of a SDS system, on the other hand, is preserved by ensuring that read/write operations can be executed in a non-blocking fashion during the reconfiguration phase, even despite the crash of (a subset of) proxy and storage nodes.

As mentioned the RM supports both per-object as well as global (valid across all objects in the SDS) changes of the quorum configurations. To simplify presentation, we first introduce the protocol for the simpler scenario of global quorum reconfigurations. We discuss how the reconfiguration protocol can be extended to support per-object granularity in Section 5.4.

5.1 Algorithm Overview

There are three different type of components involved in the execution of the reconfiguration algorithm: the storage nodes, the proxy nodes, and the RM. The purpose of the reconfiguration algorithm is to change the quorum configuration, i.e., the size of the read write quorums, used by the proxy servers.

The algorithm is coordinated by the RM. When the RM runs the configuration algorithm we say that the RM *installs* a new quorum system. We denote the quorum system being used when the reconfiguration is started as the *old quorum* and the quorum system that is installed when the reconfiguration is concluded the *new quorum*. Old and new write and read quorums are denoted, respectively, as *oldW*, *oldR*, *newW*, and *newR*. Each quorum is associated with an *epoch number*, a sequentially serial number that is incremented by the RM when some proxy is suspected to have crashed during a reconfiguration. We also assume that storage nodes maintain a variable, called *currentEpoch*, which stores the epoch number of the last quorum that has been installed by the RM. As it will be explained below, during the execution of the reconfiguration algorithm, proxy nodes use a special *transition quorum*, that is sized to guarantee intersection with both the old and new quorums.

We assume a fail stop-model (no recovery) for proxies and storage nodes, and that at least one proxy is correct. As for the storage nodes, in order to ensure the termination of read and write operations in the new and old quorum configuration, it is necessary to assume that the number of correct replicas is at least *max(oldR, oldW, newR, newW)*. For ease of presentation, we assume that the sets Σ and Π are static, i.e. nodes are not added to these sets, nor are they removed even after a crash. In order to cope with dynamic groups, one may use group membership techniques, e.g. [4], which are orthogonal to this work.

```
int epNo=0; // Epoch identifier
int cfNo=0; // Configuration round identifier
int curR=1, curW=N; // Sizes of the read and write quorums
// Any initialization value s.t. curR+curW>N is acceptable.
changeConfiguration(int newR, int newW)
    wait canReconfig;
    canReconfig = FALSE;
    cfNo++;
    broadcast [NEWQ, epNo, cfNo, newR, newW ] to $\Pi$;
    $\forall p_i \in \Pi$ :
      wait received [ACKNEWQ, epNo] from $p_i \vee suspect(p_i)$;
    if $\exists p_i : suspect(p_i)$ then
        tranR=max(curR,newR); tranW=max(curW,newW);
        epochChange(max(curR,curW),tranR,tranW);
    broadcast [CONFIRM, epNo, newR, newW ] to $\Pi$;
    $\forall p_i \in \Pi$ :
      wait received [ACKCONFIRM, epNo] from $p_i \vee suspect(p_i)$;
    if $\exists p_i : suspect(p_i)$ then
        epochChange(max(newR,newW),newR,newW);
    curR=newR; curW=newW;
    canReconfig = TRUE;
epochChange(int epochQ, int newR, int newW)
    epNo++;
    broadcast [NEWEP,epNo,cfNo,newR, newW ] to $\Sigma$;
    wait received [ACKNEWEP, epNo] from epochQ $s_i \in \Sigma$;
```
Algorithm 2: Reconfiguration Manager pseudo-code.

```
int lEpNo=0; // Epoch identifier
int lCfNo=0; // Configuration round identifier
set Q={}; // list of cfNo along with respective read/write quorum sizes
int curR=1, curW=N; // Sizes of the read and write quorums
// Any initialization value s.t. curR+curW>N is acceptable.
upon received [NEWQ, epNo, cfNo, newR, newW ] from RM
    if $lEpNo \leq$ epNo then
        lEpNo=epNo;
        lCfNo=cfNo;
        Q=Q $\cup$ < cfNo, newR, newW > ;
        int oldR=curR; int oldW=curW;
        // new read/writes processed using transition quorum
        tranR=max(oldR,newR); tranW=max(oldW,newW);
        wait until all pending reads/writes issued using the old quorum complete;
        send [ACKNEWQ, epNo ] to RM;
upon received [CONFIRM, epNo, newR, newW ] from RM
    if $lEpNo \leq$ epNo then
        lEpNo=epNo;
        curR=newR; curW=newW;
        send [ACKCONFIRM, epNo ] to RM;
```
Algorithm 3: Proxy pseudo-code (quorum reconfiguration).

We assume that the RM never fails — as mentioned before, standard techniques may be used to derive a fault-tolerant implementation of the RM — and that it is equipped with an eventually perfect failure detection service [6] that provides, possibly erroneous, indications on whether any of the proxy nodes has crashed. An eventually perfect failure detector ensures *strong completeness*, i.e., all faulty proxy processes are eventually suspected, and *eventual strong accuracy*, i.e., there is a time after which no correct proxy process is ever suspected by the RM. The reconfiguration algorithm is indulgent [17], in the sense that in presence of false failure suspicions only the liveness of read/write operations can be endangered (as we will see they may be forced to re-execute), but neither the safety of the quorum system (i.e., the Dynamic Quorum Consistency property), nor the termination of the reconfiguration phase can be compromised by the occurrence of false failure suspicions. The failure detection service is encapsulated in the *suspect* primitive, which takes as input a process identifier $p_i \in \Pi$ and returns true or false depending on whether p_i is suspected to have crashed or not. Note that proxy servers are not required to detect the failure of storage servers nor vice-versa.

In absence of faults, the RM executes a two-phase reconfiguration protocol with the proxy servers, which can be roughly summarized as follows. In the first phase, the RM informs all proxies that a reconfiguration must be executed and instructs them to i) start using the transition quorum instead of the old quorum, and ii) wait till all the pending operations issued using the old quorum have completed. When all proxies reply, the RM starts the second phase, in which it informs all proxies that it is safe to start using the new quorum configuration.

This mechanism guarantees that the quorums used by read/write operations issued concurrently to the quorum reconfiguration intersect. However, one needs to address also the scenario in which a read operation is issued on an object that was last written in one of the previous quorum configurations, i.e., before the installation of the current quorum. In fact, if an object were to be last written using a write quorum, say *oldW*, smaller than the one used in the current configuration, then the current read quorum may not intersect with *oldW*. Hence, an obsolete version may be returned, violating safety. We detect this scenario by storing along with the object's metadata also a logical timestamp, $cfNo$, that identifies the quorum configuration used when the object was last written. If the version returned using the current read quorum was created in a previous quorum configuration having identifier $cfNo$, the proxy repeats the read using the largest read quorum used in any configuration installed since $cfNo$ (in case such read quorum is larger than the current one).

Since failure detection is not perfect, in order to ensure liveness the two-phase quorum reconfiguration protocol has to advance even if it cannot be guaranteed that all proxies have updated their quorum configuration. To this end, the RM triggers an epoch change on the back-end storage nodes, in order to guarantee that the operations issued by any unresponsive proxy (which may be using an outdated quorum configuration) are preventively discarded to preserve safety.

5.2 Quorum Reconfiguration Algorithm

The pseudo code for the reconfiguration algorithm executed at the Replication Manager side is depicted in Algorithm 2. The reconfiguration can be triggered by either the Autonomic Manager, or by a human system administrator, by invoking the *changeConfiguration* method and passing as arguments the new sizes for the read and write quorums, *newQ* and *WriteQ*. Multiple reconfigurations are executed in sequence: a new reconfiguration is only started by the RM after the previous reconfiguration concludes.

Failure-free scenario. To start a reconfiguration, the RM broadcasts a **NEWQ** message to all proxy nodes. Next, the RM waits till it has received an **ACKNEWQ** message from every proxy that is not suspected to have crashed.

Upon receipt of a **NEWQ** message, see Algorithm 3, a proxy changes the quorum configuration used for its future read/write operations by using a *transition quorum*, whose read, respectively write, quorum size is equal to the maximum of the read, respectively write, quorum size in the old and new configurations. This ensures that the transition read (tranR), resp. write (tranW), quorum intersects with the write, resp. read, quorums of both the old and new configurations. Before replying back to the RM with an **AckNewQ** message, the proxy waits until any "pending" operations it had issued using the old quorum completed.

If no proxy is suspected to have crashed, a **Confirm** message is broadcast to the proxy processes, in order to instruct them to switch to the new quorum configuration. Next, the RM waits for a **AckConfirm** reply from all the non-suspected proxy nodes. Finally, it flags that the reconfiguration has finished, which allow for accepting new reconfigurations requests.

The pseudo-code for the management of read and write operations at the proxy nodes is shown in Alg. 4 and Alg. 5. As already mentioned, in case a read operation is issued, the proxies need to check whether the version returned using the current read quorum was created by a write that used a write quorum smaller than the one currently in use. To this end, proxies maintain a set Q containing all quorum configurations installed so far[2] by the Autonomic Manager. If the version returned using the current read quorum was created in configuration $cfNo$, the proxy uses set Q to determine the value of the largest read quorum used in any configuration since $cfNo$ till the current one. If this read quorum, noted oldR (see line 17 of Alg. 4), is larger than the current one, the read is repeated using oldR. Further, the value is written back using the current (larger) write quorum. Note that re-writing the object is not necessary for correctness. This write can be performed asynchronously, after returning the result to the client, and is meant to spare the cost of using a larger read quorum when serving future reads for the same object.

Coping with failure suspicions. In case the RM suspects some proxy while waiting for an **AckNewQ** or an **AckConfirm** message, the RM ensures that any operation running with an obsolete configurations is prevented from completing. To this end, the RM relies on the notion of *epochs*. Epochs are uniquely identified and totally ordered using a scalar timestamp, which is incremented by the RM whenever it suspects the failure of a proxy at lines 11 and 16 of Alg. 2. In this case, after increasing the epoch number, the RM broadcasts the **NewEp** message to the storage nodes. This message includes 1) the new epoch identifier, and 2) the configuration of the transition quorum or of the new quorum, depending on whether the epoch change was triggered at the end of the first or of the second phase.

Next, the RM waits for acknowledgements from an *epoch-change quorum*, whose size is determined in order to guarantee that it intersects with the read and write quorums of any of the configurations in which the proxies may be executing. Specifically, if the epoch change is triggered at the end of the first phase, the size of the epoch-change quorum is set equal to the maximum between the size of the read and write quorums in the old configuration. It is instead set equal to the maximum between the size of the read and write quorums of the new configuration, if the epoch change is triggered at the end of the second phase. As we shall discuss more in detail in Section 5.3, this guarantees that the epoch change quorum intersects with the read and write quorums of any operation issued by nodes that may be lagging behind, and not have updated their quorum configuration yet.

When a storage node (see Alg. 6) receives an **NewEp** message tagged with an epoch identifier larger than its local epoch timestamp, it updates its local timestamp and *rejects* any future write/read operation tagged with a lower epoch timestamp. It then replies back to the RM with an **AckNewEp** message. Whenever an an operation issued by a proxy in an old epoch is rejected, the storage node does not process the operation and replies with a **Nack** message, in which it specifies the current epoch number and the quorum configuration of this epoch.

```
1  upon received [Read, oId] from client $c$
2      while $true$ do
3          broadcast [Read, oId, curEpNo] to $\Sigma$;
4          wait received [ReadReply, oId, val, ts, W] from
           $\Sigma' \subseteq \Sigma$ s.t. $|\Sigma'|$=curR $\vee$ ([Nack, epNo,newR, newW ]
           $\wedge$epNo $> lEpNo$);
5          if received [Nack, epNo, cfNo, newR, newW ] then
6              lEpNo=epNo; lCfNo=cfNo;
               curR=newR; curW=newW;
7              Q=Q $\cup <$ cfNo, newR, newW $>$ ;
8              continue; // re-transmit in the new epoch
9          $v$= select the value with the freshest timestamp;
10         // Set of read quorums since v.cfNo till lCfNo;
11         $S = \{R_i :< q_i, R_i, \cdot > \in$Q $\wedge v.cfNo \leq q_i \leq lCfNo\}$;
12         if $max(S) \leq curR$ then
13             // safe to use cur. read quorum
14             send [ReadReply, oId, v] to client $c$;
15         else
16             // compute read quorum when v was created.
17             int oldR=max(S);
18             // obtain a total of oldR replies.
19             wait received [ReadReply, oId, val, ts] from
               $\Sigma' \subseteq \Sigma$ s.t. $|\Sigma'|$=oldR $\vee$ ( [Nack, epNo,newR,
               newW ] $\wedge$epNo $> lEpNo$);
20             if received [Nack, epNo, cfNo, newR, newW ]
               then
21                 lEpNo=epNo; lCfNo=cfNo;
                   curR=newR; curW=newW;
22                 Q=Q $\cup <$ cfNo, newR, newW $>$ ;
23                 continue; // re-transmit in the new epoch
24             $v$= select the value with the freshest timestamp;
25             send [ReadReply, oId, v] to client $c$;
26             // write v using the current quorum
27             write($v$,$oId$,$v.ts$);
28         break;
29     end
```

Algorithm 4: Proxy pseudo-code (read logic).

Upon receiving a **Nack** message (see Algs. 4 and 5), the proxy node is informed of the existence of a newer epoch, along with the associated quorum configuration. Hence, it accordingly updates its local knowledge (i.e., its epoch and read/write quorum sizes), and re-executes the operation using the new epoch number and the updated quorum configuration.

5.3 Correctness Arguments

Safety. We need to show that the quorum used by a read operation intersects with the write quorum of any concurrent write operation, and, if no concurrent write operation exists, that a read quorum intersects with the quorum used by the last completed write operation. We denote with *oldR, oldW*, *newR, newW* and *tranR, tranW* the read and write quorums used, respectively in the initial, new and transition phase.

As already mentioned, since *tranR=max(oldR,newR)* and *tranW =max(oldW,newW)*, the read, resp. write, quorums used during the transition phase intersect necessarily with the write, resp. read, quorums in both the new and old phases. Hence, safety is preserved for operations issued using the transition quorums.

In absence of failure suspicions, if any proxy process starts using the new quorum, the protocol guarantees that there is no pending concurrent operation running with the old quorum. It remains to discuss the case in which a read operation uses the new quorum size and there are no concurrent write operations. If the version returned using the current read quorum, say v, was created in the

[2]In practice, the set Q can be immediately pruned whenever the maximum read quorum is installed.

```
1  upon received [Write, oId, value] from client c
2      write (val, oId, getTimestamp());
3      send [WriteReply, oId] to client c;
4  write(value v, objId oid, timestamp ts)
5      while true do
6          broadcast [Write, oId, val, ts, curEpNo] to $\Sigma$;
7          wait received [WriteReply, oId] from $\Sigma' \subseteq \Sigma$ s.t.
           $|\Sigma'|$=curW $\vee$ ([NACK, epNo,newR, newW ] $\wedge$epNo $> lEpNo$);
8          if received [NACK, epNo, cfNo, newR, newW ] then
9              lEpNo=epNo; lCfNo=cfNo;
               curR=newR; curW=newW;
10             Q=Q $\cup <$ cfNo, newR, newW $>$ ;
11             continue; // re-transmit in the new epoch
12         break;
13     end
```
Algorithm 5: Proxy pseudo-code (write logic).

```
1  int lEpNo=0; // Epoch identifier
2  int lCfNo=0; // Configuration round identifier
3  int curR=1, curW=N; //Sizes of the read and write quorums
4  // Any initialization value s.t. curR+curW>N is acceptable.
5  upon received [NEWEP,epNo, cfNo, newR, newW ] from
   RM
6      if epNo $\geq lEpNo$ then
7          lEpNo=epNo;
8          lCfNo=cfNo;
9          curR=newR; curW=newW;
10         send [ACKNEWEP, epNo] to Reconfiguration
           Manager;
11 upon received [Read, epNo, ...] or [Write, epNo, ...] from
   $\pi_i \in \Pi$
12     if epNo $<lEpNo$ then
13         send [NACK, epNo, cfNo, newR, newW ] to $p_i$;
14     else
15         process read/write operation normally;
16         if operation is a write then
17             store lCfNo in the version metadata cfNo;
18         else
19             piggyback cfNo to the ReadReply message;
20         end
21     end
```
Algorithm 6: Storage node pseudo-code.

current quorum configuration, then the last created version is necessarily returned. Let us now analyze the case in which the last write was performed using a previous quorum configuration with timestamp $cfNo$. In this case the proxy repeats the read with the largest read quorum of any configuration installed since $cfNo$. This read quorum is guaranteed to intersect with any write operation issued since the creation of v. Hence, the read is guaranteed to return the latest value written by any non-concurrent write operation.

If the RM suspects a node, either at the end of first or of the second phase of the reconfiguration protocol, an epoch change is triggered. This ensures that the storage nodes commit to reject the operations issued in the previous epoch. The values of epochQ are chosen large enough to guarantee intersection of the epoch change quorum with the quorum used by any operation of a node that may be lagging behind, and may not have updated his quorum configuration as requested by the RM. This in its turn guarantees that the operation will gather at least a NACK, and will be re-executed in the new epoch. At the end of phase 1 epochQ is set to max(*oldR,oldW*), as the proxy may be executing using either the old quorum or the transition quorum. In fact, the transition quorum is guaranteed to intersect with a quorum of size epocQ=max(*oldR,oldW*) by construction; also, a quorum of size epocQ=max(oldR,oldW) necessarily intersects with both oldR and oldW. Since at the end of phase 2 epochQ, the proxies may be using either the new quorum or the transition quorum, the epoch change quorum is set to max(*newR, newW*) for analogous reasons.

Liveness. We start by showing that if a **changeConfiguration** is triggered at the RM, it eventually terminates.

In absence of failure suspicions, the first and second phase of the reconfiguration protocol complete since: i) we assume reliable channels; ii) the wait conditions at lines 11 and 16 of Alg. 2 eventually complete since we assume that the number of correct storage nodes is max(*oldR,oldW,newR,newW*); iii) no other blocking primitives are executed.

By the strong completeness accuracy of failure detection, if a process is faulty, it will be eventually suspected. This ensures that the waits at lines 12 and 17 of Alg. 2 will eventually unblock if some proxy fails. This is true clearly even if some proxy is falsely suspected to have crashed. In either case, an epoch change will be triggered at the end of phase 1 and/or phase 2. The epoch change phase is non-blocking, given our assumption on the number of correct storage nodes.

Let us now show that if a correct proxy node p_i receives a NEWQ message with a new quorum configuration *newR, newW*, it eventually installs the new configuration. As discussed above, the corresponding instance of the reconfiguration protocol eventually terminates, possibly triggering an epoch change. If no epoch change is triggered, eventually p_i receives a CONFIRM message by the RM and installs the new quorum. If an epoch change is triggered, there are 2 cases: i) p_i can either receive the CONFIRM message by the RM, or ii) it receives a NACK from a storage node while executing a read or write operation. In both cases, p_i installs the new quorum (line 19 of Alg. 3, lines 6 and 20 of Alg. 4, and line 10 of Alg. 5).

Finally let us analyze the termination of read and write operations. Since we are assuming the availability of both the old and new quorum configurations, in absence of failure suspicions/epoch changes the read/write quorums are eventually gathered. Also the read/write quorums do not contain any NACK message and the read/write operations complete without retrying. In presence of failure suspicions, the reconfiguration protocol can trigger up to two epoch changes that may cause pending operations to be re-executed twice. However, at each re-execution, upon receiving a NACK message (lines 4-5 of Alg. 4 and 8-9 of Alg. 5), the proxy learns the new quorum configuration and updates it epoch number. In order to ensure the eventual successful termination of read/write operations, we need therefore to assume either that i) the faulty nodes are eventually removed from the system, which eventually causes the RM not to suspect any proxy, or that ii) the time interval between two subsequent quorum reconfigurations is sufficiently long to allow operations to execute successfully in the most recent epoch, possibly after a finite number of re-executions.

5.4 Per-object Quorum Reconfiguration

As discussed, the above presented protocol allows for altering the quorum size used by the entire data store. However, extending the above presented protocol to allow for tuning independently the quorum sizes used to access different objects in the data store is relatively straightforward.

In particular, during the initial, round based optimization phase described in Section 4, the RM is provided with a set of object identifiers and with their corresponding new quorum configurations. The RM forwards this information to the proxies via the *NewQ* message. The proxy servers, in their turn, shall store the mapping between the specified object identifiers and the corresponding write quorums, and use this information whenever they are serving a read or a write operation. Note that, in our prototype, we store this mapping in main memory, as only a reduced set of "hotspots" is individually optimized. This simple mechanism allows the proxy servers to determine the new and old quorum sizes to use on a per object basis. Also, when a fine-grain quorum reconfiguration is requested that only affects a set of data items D, the proxy should wait only, in the first phase of the reconfiguration algorithm, for the completion of any pending operation (using the old quorum) targeting some of the items in D.

Any (read/write) access to objects whose quorum size has not been individually optimized can use a common, global quorum configuration and be treated exactly as shown in Section 5.2.

6. THE ORACLE

The Oracle is responsible for determining the best quorum configuration for the current workload conditions. We cast the selection of the optimal quorum configuration as a classification problem [29], in which one is provided with a set of input metrics (also called features) describing the current state of the system and is required to determine, as output, a value from a discrete domain (i.e., the best performing quorum configuration among a finite set in our case). In Q-OPT we rely on black-box machine learning techniques to automatically infer a predictive model of the optimal quorum configuration.

To this end, Q-OPT exploits the C5.0 algorithm [34]. C5.0 builds a decision-tree classification model in an initial, off-line training phase during which a greedy heuristic is used to partition, at each level of the tree, the training dataset according to the input feature that maximizes information gain. The output model is a decision-tree that classifies the training cases according to a compact (human-readable) rule-set, which can then be used to classify (decide the best quorum strategy) future scenarios.

The accuracy achievable by any machine learning technique is well known to be strictly dependant on the selection of appropriate input features [29]. In Q-OPT we selected a restricted set of workload characterization metrics $\vec{\mathscr{W}}$ that can be measured using lightweight, non-intrusive monitoring mechanisms, and which capture both the proxies' and the storage nodes' current capacity to serve each type of request. The input features include: number of read and write operation requests received, execution time of read and write operations, number of read and write operations that were successfully served, percentage of read operations, average sizes of the read and written objects, number of proxy and storage nodes.

In order to build a training set for the classification model, Q-OPT relies on an off-line training phase during which we test a heterogeneous set of synthetic workloads and measure a reference KPI of the system (e.g., throughput or response time) when using different read/write quorum configurations. The training set is then composed by tuples of the form $\langle curW, \vec{\mathscr{W}}, bestW\rangle$, where $curW$ is the write quorum configuration used while executing workload $\vec{\mathscr{W}}$. $bestW$ is the best performing write quorum configuration for $\vec{\mathscr{W}}$ and the target class for the classification problem.

This model construction methodology allows us to accommodate in a simple way also for application-dependant constraints on the quorum sizes, e.g. imposing a minimum value for the size for the write quorums for fault-tolerance purposes. To this end, it suffices to restrict the tuples in the training set to include only admissible values of both $curW$ and $bestW$. More in detail, tuples using illegal values of $curW$ are discarded from the training set. Tuples that use legal values of $curW$ and whose optimal quorum configuration, $bestW$, violates some application-dependant constraint are retained in the training set, but their $bestW$ value is replaced with the best performing among the *feasible* write quorum configurations.

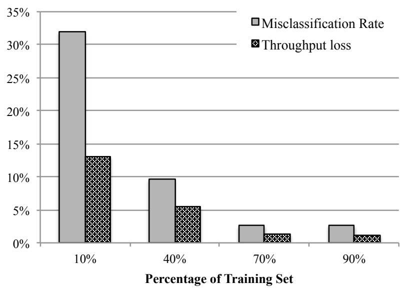

Figure 5: Oracle's misclassification rate and % throughput loss.

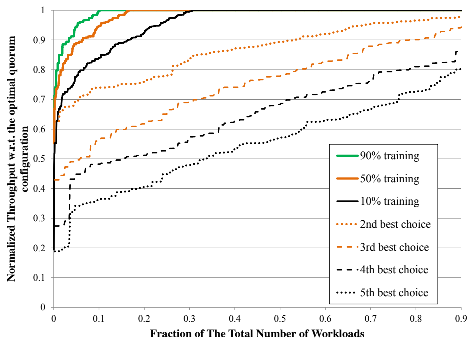

Figure 6: Cumulative distribution of the Oracle's accuracy.

7. EVALUATION

This section presents the results of an experimental study aimed at assessing three main aspects: the accuracy of the ML-based Oracle; the effectiveness of Q-OPT in automatically tuning the quorum configuration in presence of complex workloads; and the efficiency of the quorum reconfiguration algorithm.

7.1 Accuracy of the Oracle

In order to assess the accuracy of the Oracle, we consider the same set of 170 workloads used in Figure 3 (Section 2). Figure 5 reports the misclassification rate and throughput loss (w.r.t. the optimal solution) when we vary the size of the training set, using the rest of available data as test set. The results are obtained as the average of 200 runs, in which we fed the C5.0 with different (and disjoint) randomly selected test and training sets. The results show that the ML-based oracle achieves very high accuracy, i.e. misclassification rate lower than 10% and throughput loss lower than 5%, if we use as little as the 40% of the collected data set as training set. Interestingly, the throughput loss is normally less than half of the misclassification rate: in fact, in most of the misclassified workloads, the quorum configuration selected by the oracle yields performance levels that are quite close to the optimum.

Figure 6 provides an alternative perspective on our data set. It reports the cumulative distribution functions of the accuracy achieved by our predictor with different training set sizes, and contrasting them with the normalized performances using all possible configurations for each considered workload. The plot highlights that the choice of the quorum configuration has a striking effect on Q-OPT's performance: in about 20% of the workloads, the selection of the third best quorum configuration is 40% slower than the optimal one; in the worst case, the plot also shows that the worst (i.e.,

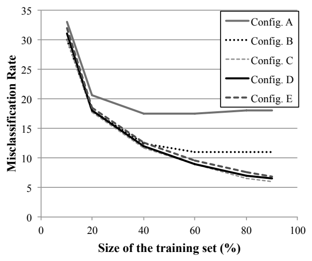

Figure 7: Oracle's misclassification rate when varying the set of features used.

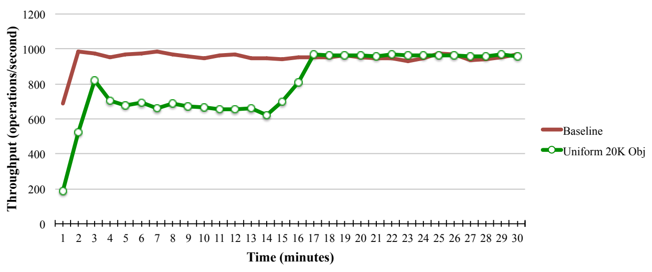

Figure 9: Evaluating the overheads associated with the bulk reconfiguration of the quorum size. Write quorum is increased from 1 to 5 in presence of a read-dominated workload.

the 5-th best) choice of the quorum configuration can be even more that 5x slower than the optimal for some workloads.

Figure 7 allows us to asses to what extent the selection of the features used to generate the ML-based models impacts the model's accuracy. We used five different configurations for this experiment, from A to E. Each configuration progressively adds extra features, including previous configuration features. As the plot shows, the configuration A, which only includes the percentage of write transactions as feature, achieves poor accuracy, generating a misclassification rate of around 35%. This result confirms the relevance of using a multi-variate model, capable of keeping into account additional factors besides the write percentage (as suggested also by the plot in Figure 2. Indeed, the plot clearly shows that, as we include among the provided features also the object size (Conf. B) and throughput (Conf. C) the misclassification rate gets considerably reduced. However, adding additional features (e.g., conf. D and E, which include statistics on the latencies perceived by get and put operations) does not benefit accuracy, but, on the contrary, can lead to overfitting [29] phenomena that can ultimately have a detrimental effect on the learner's accuracy.

Finally, in Table 1, we report the accuracy achieved by the considered learner when using the, so called, *boosting* technique. The boosting approach consists in training a chain of N learners, where the learner in position i is trained to learn how to correct the errors produced by the chain of learners in position $1,2,\ldots,i$. This technique has been frequently reported to yield significant accuracy improvements when used with weak learners. Our experiments do confirm the benefits of this technique, although the relative gains in accuracy are, at least for the considered data set, not so relevant to justify its additional computational overheads.

7.2 Reconfiguration Overhead

Q-OPT uses a two-phase quorum reconfiguration protocol whose latency is affected by the number of pending operations each proxy has to finish before completing the first phase of the protocol. Thus, we expect to observe an increase in latency as the number of clients issuing concurrent operations increases. Figure 8 shows the quorum reconfiguration latency in absence of faults varying the number of clients from 15 to 150 (i.e., close to system's saturation that we estimate at around 165 clients). In this and in the following experiments we used object sizes uniformly distributed in the [10KB - 50KB] range. As expected, the results show a correlation between the latency of the reconfiguration and the request's arrival rate; however, the reconfiguration latency remains, in all the tested scenarios, lower than 15 milliseconds, which confirms the efficiency of the proposed reconfiguration strategy.

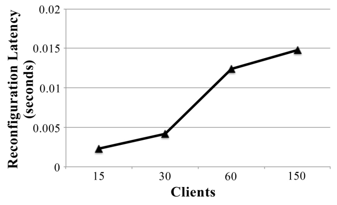

Figure 8: Reconfiguration latency vs system load.

Finally, Figure 9 focuses on evaluating the performance of the lazy write-back procedure encompassed by the quorum reconfiguration protocol. Recall that this happens when a read operation gathers a quorum of replies that reveals that the last write applied to the read object has been performed using a smaller write quorum than the one currently used. In this case, the read operation is forced to wait for additional replies before returning to the client, and it has to write back the data item using the new (larger) write quorum. The experiment whose results are shown in Figure 9 is focused precisely on evaluating the overhead associated with these additional writes. To this end we considered a worst-case scenario in which: i) the system is heavily loaded, ii) data items are accessed with a uniform distribution, and iii) the write quorum increases from 1 to 5 while the application is running a read-dominated (95%) workload. At the beginning of the experiment we trigger the bulk reconfiguration (i.e., for the entire set of 20K data items in the SDS) of the quorum and report throughput over time. The plot shows that, even in such a worst case scenario, the throughput loss due to the additional writes performed when reading an object for the first time using the new quorum is limited to around 12%. Indeed, we argue that it would be possible to further reduce the overheads due to this write back process by reducing the frequency with which write backs are issued and batching writes to amortize their cost. This would imply a trade-off between the time to complete the reconfiguration phase and the overhead incurred while this is in progress.

7.3 System Performance

Finally, Figure 10 evaluates the effectiveness of Q-OPT over time when faced with complex workloads. We consider the following baseline configurations. *R1W5*, *R3W3* and *R1W5* are static configurations that force the system to use the same quorum for all the objects. *AllBest* uses the optimal quorum for each of the objects. Finally, *Top10%* uses the optimal quorum for each of the 10% most accessed objects. Notice that *AllBest* and *Top10%* are unachievable configurations in practice since it would require precise pre-knowledge about the workloads of each object.

In the experiment we combine two workloads, a read intensive (95% of reads) and a write intensive (95% of writes) one, each representing a different tenant. This means that each workload ac-

	10% Training		50% Training		90% Training	
	unboosted	boosted	unboosted	boosted	unboosted	boosted
Avg. Misclassification (%)	15	14	9	7	6	5
Avg. Distance from optimal (%)	4.6	4	2.2	1.6	1.6	0.9
Avg. Distance when misclassified (%)	14.1	12.9	10.9	9.5	9.5	7.6

Table 1: Impact of boosting in the Oracle's performance when varying the training set size.

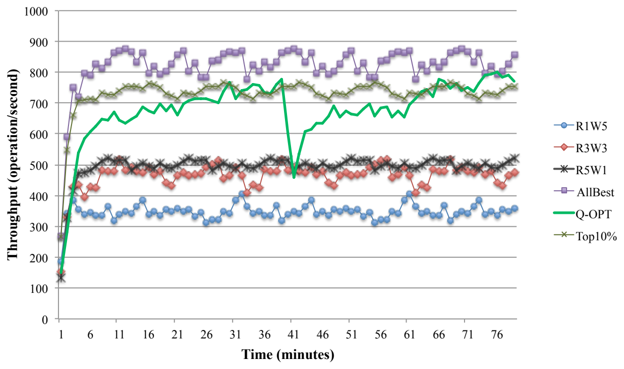

Figure 10: Q-OPT performance in comparison to other configurations.

cesses a non-overlapping set of objects. After 40 minutes, we swap the type of workload. Therefore, the read intensive workload becomes write intensive and viceversa. The idea is to observe how Q-OPT reacts to changes in the workloads. For this experiment, Q-OPT runs a fine-grain optimization round every minute. After 20 minutes, the fine-grain optimization phase ends and Q-OPT optimizes the remaining objects in bulk (i.e., using a single, homogeneous quorum configuration). After the swapping (minute 40), Q-OPT starts again the fine-grain optimization phase and continues behaving as described for the first 40 minutes.

Q-OPT behaves as expected. During the first 20 minutes the throughput grows as the fine-grain optimization rounds advance, getting close to the *Top10%* baseline right before optimizing the tail. Once the tail is optimized (after minute 20), the throughput keeps growing even beyond the *Top10%* line and getting closer to the *AllBest* configuration. The *Top10%* configuration does not optimize the tail; therefore, it is expected that Q-OPT outperforms it, at least slightly. As expected, the plot shows that the performance of Q-OPT matches closely that of the optimal configurations. These results confirm the accuracy of Q-OPT's Oracle and highlight that the overheads introduced by the supports for adaptivity are very reduced. After swapping the workloads, Q-OPT experiences a noticeable decrement in the performance since it has to start the optimization from scratch.

Furthermore, the figure shows that none of the static configurations achieves a throughput comparable to Q-OPT's. In the worst case, the *R1W5* configuration is more than 2x slower than Q-OPT during stable periods. Even for the best static configuration (*R5W1*), Q-OPT still achieves around 45% higher throughput.

8. RELATED WORK

Several works have explored the idea of selectively relaxing consistency in cloud storages[36, 7, 23, 3] in order to minimize the probability of violating SLAs defined on KPIs like throughput or latency. Techniques include dynamically selecting which servers should serve incoming requests[36], sizing quorums in order to ensure probabilistic bounded staleness[3] and adapting the concurrency control mechanism to the criticality of managed data[3]. Unlike these solutions, Q-OPT aims at ensuring strong consistency (by enforcing strictness of the quorum system used to replicate data) at the minimum cost, i.e. by automatically identifying the optimal quorum configuration given the current application's workload.

Our work is also related to the literature on the specification of adaptation policies [35, 2] which provide an infrastructure to allow experts (such as programmers or system administrators) to control the adaptation policies of a complex system by means of different types of rules' system, and on ML techniques to automate the determination of the adaptation policy [12, 37, 31]. Q-OPT falls in the latter class of systems, and relies on decision-tree classifiers to infer automatically the adaptation policy of a strict quorum system.

Q-OPT has relations also with our previous works[10, 9] in which we tackled the problem of dynamically adapting the replication protocol used by in-memory transactional systems, also exploiting black-box machine learning techniques. However, quorum-based distributed storage systems have significant differences with respect to the platforms considered in our prior work, e.g. partial vs full replication, in-memory vs durable storage, single object operations vs transactions. As such, Q-OPT needs not only to cope with different algorithmic challenges to support reconfiguration. It also requires to reformulate the ML-based optimization problem, e.g. to account for the existence of constraints on the minimum number of replicas of each object.

The quorum reconfiguration algorithm integrated in Q-OPT has relations with the vast body of literature on dynamic quorum systems[13, 19] and of reconfiguration of atomic storage systems[16, 1]. These works target a different, and more complex problem that the one tackled by the reconfiguration algorithm proposed in Section 5: they allow to redefine at run-time *which* nodes should be included in a quorum system in a dynamic environments in which nodes may join or leave the system. On the other hand, the concern of our proposal is to enforce agreement among the set of proxy nodes (which is typically disjoint from the set of storage nodes) on the sizes of the read and write quorums to use when interacting with the storage nodes. A key difference with respect to these techniques is that our work decouples the problem of group membership (delegating it to some external service[18]) from that of quorum optimization (the focus of our work). Not only this separation of concerns leads to a significant reduction of complexity, it allows also for modularly deploying our solution on off-the-shelf SDS/cloud storage systems (like Swift or Cassandra) in a non-intrusive fashion, i.e. with minimal changes to the existing infrastructure.

9. CONCLUSIONS

This paper tackled the problem of automating the tuning of read/write quorum configuration in distributed storage systems, a problem that is particularly relevant given the emergence of the software defined storage paradigm. The proposed solution, which we called Q-OPT, leverages ML techniques to automate the identification of the optimal quorum configurations given the current application's workload, and on a reconfiguration mechanism that allows non-blocking processing even during quorum reconfigurations. Q-OPT's optimization phase first focuses on optimizing the most accessed objects in a fine-grain manner. Then, it ends by assigning the same quorum for all the objects in the tail of the access distribution based on their aggregated profile. We integrated Q-OPT in a popular, open-source software defined storage and conducted an extensive experimental evaluation, which highlighted both the accuracy of its ML-based predictive model and the efficiency of its quorum reconfiguration algorithm.

10. REFERENCES

[1] M. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Proc. PODC 2009*, pages 17–25. ACM, 2009.

[2] R. Bahati, M. Bauer, and E. Vieira. Policy-driven autonomic management of multi-component systems. In *Proc. CASCON '07*, pages 137–151. IBM Corp., 2007.

[3] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, Apr. 2012.

[4] K. Birman and R. V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. 1994.

[5] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. *The primary-backup approach*, pages 199–216. ACM Press/Addison-Wesley, 1993.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[7] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *Proc. CLUSTER 2012*, pages 293–301. IEEE Computer Society, 2012.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. SOCC 2010*, pages 143–154. ACM, 2010.

[9] M. Couceiro, P. Romano, and L. Rodrigues. Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In *Proc. Middleware 2011*, pages 309–328. Springer-Verlag, 2011.

[10] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. In *Proc. DSN 2013*, pages 1–12, 2013.

[11] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP 2007*, pages 205–220. ACM, 2007.

[12] D. Didona, P. Romano, S. Peluso, and F. Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proc. ICAC 2012*, 2012.

[13] D. Dolev, I. Keidar, and E. Y. Lotem. Dynamic voting for consistent primary components. In *Proc. PODC 1997*, pages 63–71, 1997.

[14] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: Understanding personal cloud storage services. In *Proc. IMC 2012*, pages 481–494. ACM, 2012.

[15] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, Oct. 1985.

[16] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, pages 1–48, 2010.

[17] R. Guerraoui. Indulgent algorithms (preliminary version). In *Proc. PODC 2000*, pages 289–297. ACM, 2000.

[18] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[19] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. Database Syst.*, 12(2):170–194, June 1987.

[20] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85–126, Oct. 2004.

[21] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Trans. Database Syst.*, 28(3):257–294, Sept. 2003.

[22] R. Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[23] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, Aug. 2009.

[24] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[26] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proc. PODC 1991*, pages 1–9. ACM, 1991.

[27] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of STOC 1997*, pages 569–578.

[28] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th ICDT*, Edinburgh,Scotland, 2005.

[29] T. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[30] M. Naor and A. Wool. The load, capacity and availability of quorum systems. In *Proc. SFCS 1994*, pages 214–225, 1994.

[31] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proc. Eurosys 2010*, pages 237–250. ACM, 2010.

[32] E. Page. Continuous inspection schemes. *Biometrika*, pages 100–115, 1954.

[33] D. Peleg and A. Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, Dec. 1995.

[34] J. Quinlan. C5.0/see5.0. `http://www.rulequest.com/see5-info.html`.

[35] L. Rosa, L. Rodrigues, A. Lopes, M. Hiltunen, and R. Schlichting. Self-management of adaptable component-based applications. *IEEE Trans. Softw. Eng.*, 39(3):403–421, Mar. 2013.

[36] D. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. SOSP 2013*, pages 309–324. ACM, 2013.

[37] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. Fortes. Fuzzy modeling based resource management for virtualized database systems. In *Proc. MASCOTS 2011*, pages 32–42. IEEE Computer Society, 2011.

[38] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proc. ICDCS 2000*, pages 464–. IEEE Computer Society, 2000.